ECE 3401 Digital Systems Design – Spring 2026

Programming Assignment 2:
Digital Design of a Single-cycle RISC-V Processor

Due March 27, 2026 @ 11:59 PM on HuskyCT

In this assignment, you will implement a single cycle per instruction implementation of a subset of RISC-V instruction set architecture. The goals of this assignment are to:

- Become familiarized with assembly level instruction processing of the RISC-V instruction set architecture
- Familiarize with modeling and simulating multi-ported memory and register file components
- Implement a Controller that drives the control signal for the design Datapath
- Implement a Datapath for the instruction processing and program counter
- Use testbench based RISC-V assembly programs to verify the correctness of the design

We will be using the RISC-V instruction set architecture (ISA) for this assignment. The **riscv-uconn** ISA is described in the riscv-uconn.pdf document.

You are given an `assembler` directory with the needed RISC-V assembler. First, navigate to the `assembler` directory and execute the **make** command to build the assembler. This step requires ***gcc*** tool for compilation.

The assembly level RISC-V programs are under the `unittests` directory inside `pa2` directory. Your second step is to navigate to the `pa2` directory and compile the provided assembly programs (*.asm files in `unittests` directory) by executing **bash assemble_all.sh** at the command line. This script will create an `assembled_tests` directory and populate it with the assembled programs (*.out files in the `assembled_tests` directory).

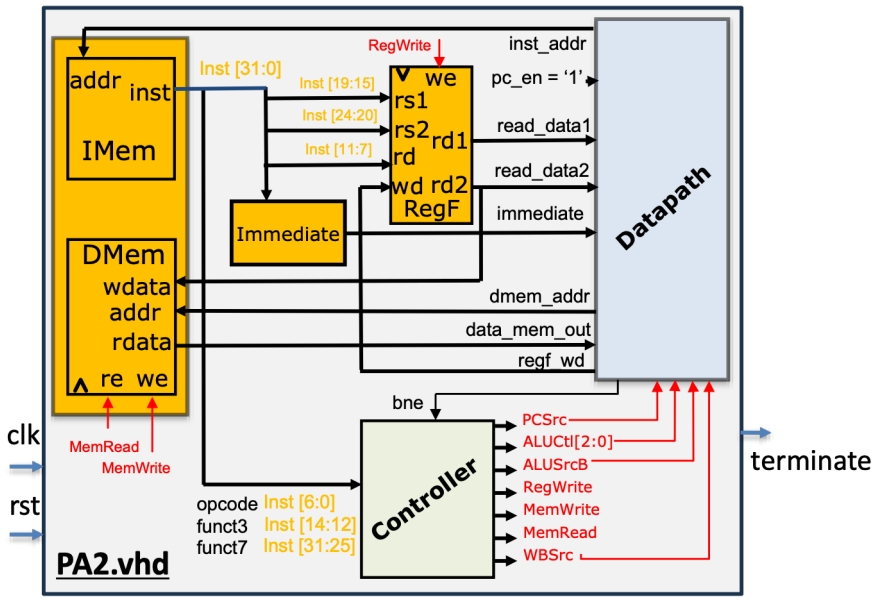
To execute one or all assembled programs using the RISC-V processor design, a **Makefile** is provided to you. To run single assembled program from the `assembled_tests` directory, just write **make <test_name>** (e.g **make array_adder**) at the command line. This will create a `Logs` directory containing the test log and a `Wave` directory containing the waveform for the test. To run all tests together, use **make dump_all** at the command line, this will create a `Logs` directory containing a single log file with results of all tests and a `Wave` directory containing waveform files for all tests.

The waveforms produced by your design simulation are the preferred method for design verification. However, we have also provided a supplemental method. The `gold_output` directory contains the golden log files for all programs in the `unittests` directory. After simulating the tests with your design, the contents of the registers and memory updates in the `Logs` directory can be compared to the golden output using the following command: **diff -w Logs/<test_name>.log gold_output/<test_name>.log**.

The `code` directory inside `pa2` directory, has the VHDL source codes under `src` directory and the testbench under the `tb` directory.

The RISC-V design's top-level is in the PA2.vhd file in the `src` directory. It instantiates the design modules that are either given to you or are incomplete and you must complete their VHDL entry for this assignment. The testbench testbench.vhd under the `tb` directory instantiates the PA2.vhd, reset the design for 60ps and creates a clock with 100ps clock period. When the reset is de-asserted, the program loaded into the memory is executed starting from the program counter at address 0. When the terminate instruction (`addi zero, zero, 1`, see riscv-uconn.pdf for more details) is executed, the PA2.vhd asserts the terminate output and simulation ends. At this point, the design creates the test and waveform logs. You will not modify the testbench. You will also not modify the VHDL codes in the `src` directory, except the **Controller.vhd** and **Datapath.vhd** files. The PA2 design is described in the next section, along with the description of what you are expected to complete for this assignment.

You will implement a subset of the RISC-V ISA (see riscv-uconn.pdf), where each instruction executes in a single cycle. The figure shows the design components instantiated in PA2.vhd.



The instruction and data memory are instantiated using the **Imem_Dmem.vhd** file (marked with orange IMem and DMem modules in the figure). You will ***not*** modify this file. It contains simulation functions to load and initialize one program test at a time in the memory defined by the *mem_array* type. It also contains the functions needed for logging the memory outputs in the `Logs` directory. The `terminate` input signal from PA2.vhd is used to initiate the logging functions. At the end of the Imem_Dmem.vhd file, three VHDL processes implement the instruction and data memory logic. The instruction memory uses the 32-bit instruction address from Datapath.vhd, reads the instruction memory and returns the 32-bit instruction output to the PA2.vhd (`Inst` in the figure). To model the data memory as a separate module, the data memory uses separate processes for reading and writing its memory locations. The data memory uses the 32-bit address input from Datapath.vhd to read the memory and returns the 32-bit data output to the Datapath.vhd. The `MemRead` input signal is used to ensure that data memory is only read when this signal is asserted by the Controller.vhd. This prevents the design from reading out of bound memory locations from memory. To support synchronous writes to data memory, the `MemWrite` input signal is used to enable writing the 32-bit data input from the Datapath.vhd on the rising edge of the clock at the 32-bit address input. The figure shows all these input and output signals to/from the Imem_Dmem.vhd module.

The RISC-V register file is instantiated using the **Register_file.vhd** file (marked with orange RegF module in the figure). You will ***not*** modify this file. It contains the `ram_type` for the thirty-two 32-bit RISC-V registers and the binding logic for the register ABI names (see riscv-uconn.pdf). It also contains the functions needed for logging the register file outputs in the `Logs` directory. The `terminate` input signal from PA2.vhd is used to initiate the logging functions. Three VHDL processes implement the two read ports and a single write port logic for the register file. The 5-bit `rs1` input from PA2.vhd reads the register file and returns the 32-bit `read_data1` output to the Datapath.vhd. Another concurrent process uses the 5-bit `rs2` input from PA2.vhd reads the register file and returns the 32-bit `read_data2` output to the Datapath.vhd. To support synchronous writes to the register file, `we` input signal is used to enable writing the 32-bit `regf_wd` input from the Datapath.vhd on the rising edge of the clock at the 5-bit `rd` input. The figure shows all these input and output signals to/from the RegF module.

The RISC-V immediate decoder is instantiated using the **Immediate.vhd** file (marked with orange Immediate module in the figure). You will ***not*** modify this file. This module receives the 32-bit instruction output from the IMem module as the `Inst` input. A process implements the immediate logic for the various instruction types and returns a 32-bit immediate output to the Datapath.vhd (`immediate` in the figure).

You are ***required*** to complete the **Controller.vhd** and the **Datapath.vhd** files for this assignment.

Datapath Implementation

The Datapath logic implements the ALU using the 3-bit `AluCtl` input from the Controller. The first 32-bit input to the ALU is the 32-bit `read_data1` input from the register file. However, the second 32-bit input uses the `ALUSrcB` input from the Controller to either use the 32-bit `read_data2` from the register file or the 32-bit `immediate` from the Decoder. Note that the second input to the ALU is used to implement the 5-bit shift amount for the shift R-type and I-type instructions. The B-type instruction also uses the ALU for its compare operation. The outcome of the comparison is used to compute the `bne` output signal. For many instructions, the Datapath must write the output of the ALU to the register file. However, for the load (LW) instruction, the output of the DMem is written to the register file. This data path logic is supported by the `WBSrc` input from the Controller.

The Datapath supports the program counter (PC) 32-bit register that synchronously updates the PC when `pc_en` input signal is asserted. The `rst` input signal must asynchronously reset the PC to 0 when asserted. You are allowed to use two separate adders (behavioral + operator) to implement the PC update logic using the 32-bit `immediate` input from the Decoder and the sequential increment of the PC value. The selection of PC is controlled using the `PCSrc` input from the Controller.

Controller Implementation

The controller logic receives the `opcode` (7-bits), `funct3` (3-bits), and `funct7` (7-bits) from the 32-bit instruction output from the IMem module and the `bne` input signal from the Datapath and must derive the logic to output the control signals as described earlier. A process implements the decoding logic for the supported RISC-V instructions and populates the 32-bit `Inst_Type` internal signal in the Controller.vhd. The PA2.vhd instantiates the instruction type package in 0itype.vhd file. The *instruction_type* defines user friendly names for the instructions supported in this assignment. Using the 32-bit `Inst_Type` signal and the `bne` input signal, the control outputs from the controller must be derived using the second process.

Design Verification

We have provided a fully functional testbench.vhd. After initialization and reset, a program test being run is executed by the design. The testbench waits for the `terminate` signal and waits for 50ps before reporting the completion of the simulation. If a test does not complete in 500ns, the testbench reports and ERROR timeout message. This process can be performed sequentially for each program in the `assembled_tests` directory.

Deliverables

Please submit the following:

1. Your modified Controller.vhd and Datapath.vhd VHDL files.
2. Submit a report PDF with block diagrams for the Datapath and Controller modules clearly showing the input and output signals and behavioral components such as registers, multiplexors, arithmetic modules etc. Provide a short description of your Datapath and Controller designs to get credit.
3. Submit a single PDF with screenshots of the output waveforms for all program tests. You must use the pa2.gtkw to generate these waveforms.

In case your design is not fully functional, you will need to schedule a code review with the TA.